Video one: introduction to classes and member function

Exercise

Let's start today with a quick exercise that helps motivate the transition we'll make from C-Style to C++ Style ADTs.

Consider the code here that creates and uses a C-style ADT, specifically the `Triangle` ADT from last time:

```
1 int main() {
2   Triangle t;
3   Triangle_init(&t, 3, 4, 5);
4   cout << Triangle_perimeter(&t) << endl;
5   Triangle_scale(&t, 2);
6   cout << t.a << endl;
7 }
```

Q1: What would happen if the programmer forgot to write line 2?

no declaration for t, compile error

Save

Instructor's Feedback

Your answer has not been saved yet!

Q2: What would happen if the programmer forgot to write line 3?

undefined behavior. triangle still exists but has junk values

Save

Instructor's Feedback

Your answer has not been saved yet!

Q3: Which parameter do all of the `Triangle` functions have in common?

first param takes in address of triangle to work with

Save

Instructor's Feedback

Your answer has not been saved yet!

Q4: What's wrong with line 6? Does the compiler give us an error here?

breaks the interface for Triangle. no error, we just shoudl be vigilant and try not to do this

Save

Instructor's Feedback

Your answer has not been saved yet!

5 Introducing Classes

- A class has both **member data** and **member functions**.

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in) { ... }

  double perimeter() const { ... }
  void scale(double s) { ... }
};

int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter();
}
```

6 Member Functions

C Style(struct) | C++ Style (class)

```
void Triangle_scale(
  Triangle *tri, double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
}
```

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }
};

int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
}
```

We had to pass the address of t1 ourselves.

7 `const` Member Functions

- The `perimeter` function shouldn't change the `Triangle` (i.e. its member variables).

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  double perimeter() const {
    return this->a + this->b + this->c;
  }
};
```

const here means the this pointer will be a pointer-to-const. The effect is that this function cannot change any member variables.

Compile error since t1 is const but scale isn't.

```
int main() {
  const Triangle t1(3, 4, 5);
  cout << t1.perimeter() << endl;
  t1.scale(2);
}
```

OK. t1 is const and perimeter to resp

8 `const` Member Functions

C Style(struct) | C++ Style (class)

```
double Triangle_perimeter(
  Triangle const *tri) {
  return tri->a +
    tri->b;
    tri->c;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  cout << Triangle_perimeter(&t1);
}
```

tri is a pointer to const

```
class Triangle {
private:
  double a;
  double b;
  double c;
public:
  double perimeter() const {
    return this->a + this->b + this->c;
  }
};

int main() {
  Triangle t1(3, 4, 5);
  cout << t1.perimeter(
}
```

this is a pointer to const

9 Member Functions

- You should reuse functionality wherever you can

C Style(struct)

```
void Triangle_shrink(Triangle *tri, double s) {
  Triangle_scale(tri, 1.0 / s);
}
```

C++ Style (class)

```
class Triangle {
public:
  void shrink(double s) {
    this->scale(1.0 / s);
  }
};
```

10 Using Members Without `this`

- Members can be referred to directly in a member function.
  - (The compiler inserts `this->` for you.)

```
class Triangle {
private:
  double a, b, c;

public:
  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }

  void shrink(double s) {
    this->scale(1.0 / s);
  }
};
```

```
class Triangle {
private:
  double a, b, c;

public:
  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }

  void shrink(double
    scale(1.0 / s);
  }
};
```

video2: member accessibility and constructors

Exercise

Consider another member function, `halfPerimeter()`, which is intended to return a value that is half of the triangle's perimeter. The (questionable) algorithm we choose for our implementation is to first shrink the triangle in half and then return its perimeter.

```
class Triangle {
private:
  double a, b, c;

public:
  double perimeter() const { ... }
  void scale(double s) { ... }
  void shrink(double s) { ... }

  double halfPerimeter() const {
    shrink(2);
    return perimeter();
  }
};

int main() {
  Triangle t1(3, 4, 5);
  cout << t1.halfPerimeter();
}
```

Q5: The lines `shrink(2);` and `return perimeter();` call member functions, but what object are they called on?

specifically t1, because that's wher ethe member function call;s started. generally, the same triangle as halfPerimeter was originally called on

Save

Instructor's Feedback

Your answer has not been saved yet!

Q6: The compiler says there's some kind of `const` error with the `shrink(2);` line. Will adding `const` to the signature of `shrink` fix the problem? (Hint: Nope. But why not?)

now shrink won't compile. internally shrink must cahnge the traingle.

Save

Instructor's Feedback

Your answer has not been saved yet!

Q7: Let's remove the `const` on `halfPerimeter()`. Now the code compiles. Are there any situations in which calling `halfPerimeter()` wouldn't compile now?

yes, if the original triangle (like t1) was itself const, now halfPermiter can't be called on it.

Q8: The call to `t1.halfPerimeter()` compiles now, but what's wrong with the code? What does this mean about using `const` and the algorithm for `halfPerimeter()`?

12 Member Accessibility

- Declare members with an access level.
  - Public: Can be used anywhere.
  - Private: Can only be used in **class scope**.

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};
```

Data members are private. Member functions are public.

Accessing a, b, c here is fine since we're inside Triangle.

```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter();

  // Die triangle! DIE!
  t1.a = -1;
}
```

Ok. These member functions are public.

Compile error! a is private and accessible

People sometimes call this visibility instead of accessibility.

13 Exercise

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  bool isSame(const Triangle &someOtherTriangle) {
    return a == someOtherTriangle.a &&
           b == someOtherTriangle.b &&
           c == someOtherTriangle.c;
  }
};

int main() {
  Triangle t1(3, 4, 5);
  Triangle t2(3, 4, 7);
  cout << t1.isSame(t2);
}
```

Question

Is it OK for `isSame` to use private member variables from some other `Triangle`?

A) Yes

B) No

14 Constructors

- Whenever you create an object of class type, a **constructor** for that class is called on the object to initialize it. **Always**[1].
- A constructor is basically a function, but you don't call it yourself – the compiler does it automatically.
- All of these use a `Triangle` constructor:

```
int main() {
  Triangle t1;
  Triangle t2(3, 4, 5);
  Triangle t3 = Triangle(3, 4, 5);
}
```

default ctor | custom ctor

1 C-style structs can be initialized with an initializer list, as we've seen before, but C++ style ADTs always use constructors.

15 Defining Constructors

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {

    // nothing to do in body
  }
};

int main() {
  Triangle t2(3, 4, 5);
}
```

Same "name" as the class.

Parameters receive arguments provided to the initializers.

Constructors are usually public.

A *member initializer list* is a special syntax for initializing members in a constructor.

16 Member Initializer Lists

- Warning! The order of initialization depends on the declaration order.
  - NOT the order of the member initializer list.

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : c(c_in), b(b_in), a(a_in) {
  }
};
```

a is initialized first, then b, then c.

This ordering is ignc

17 Multiple Constructors

- A class may have several different constructors.

```
class Triangle {
private:
  double a, b, c;

public:
  Triangle()
    : a(1), b(1), c(1) { }

  Triangle(double side)
    : a(side), b(side), c(side) { }

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) { }
};
```

```
int main() {
  Triangle t1;
  Triangle t2(10);
  Triangle t3(3, 4, 5);

  Triangle t4(3, 4);
}
```

A "default" constructor.

Error: No matching constructor.

Video 3: composing C++ ADT(classes as members)

19 Classes as Members

- Members are default-initialized if left out of the member-initializer list for a constructor.

```
class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;

public:
  Professor(const string &name)
    : name(name), favCoffee(0, 0, false) {
  }
};
```

vector default ctor creates an empty vector

Triangle default ctor creates a 1x1x1 triangle

favCoffee initialized using the coffee ctor v 0 cream/sugar, not decaf

Exercise

Here again is the `Professor` class from the video:

```
class Coffee {
public:
  Coffee(int creams, int sugars);
  Coffee(int creams, int sugars,
         bool isDecaf);
};
class Triangle {
public:
  Triangle();
  Triangle(double side);
  Triangle(double a_in, double b_in,
           double c_in);
};

class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;
  ...
};
```

Consider several possible constructors for the `Professor` class. Which compile successfully? For those that don't compile, explain why (including which member is not initialized correctly).

Q10:

```
Professor(const string &name)
  : name(name) { }
```

error, missing initialization for favCoffee. no default ctor for Coffee

Save

Instructor's Feedback

Your answer has not been saved yet!

Q11:

```
Professor(int creams, int sugars)
  : favCoffee(creams, sugars) { }
```

ok. acceptible default initialization for name, students, and favTriangle

Save

Instructor's Feedback

Your answer has not been saved yet!

Q12:

```
Professor(const string &name,
          const string &student)
  : name(name) {
  students.push_back(student);
}
```

error, missing initialization for favCoffee. no default ctor for Coffee

Save

Instructor's Feedback

Your answer has not been saved yet!

Q13:

```
Professor(const Coffee &coffee)
  : name("Laura"),
    favCoffee(coffee),
    favTriangle(3, 5) { }
```

error - initializaation of favTriangle is improper, no ctor for Triangle that takes 2 ints

Save

Instructor's Feedback

Your answer has not been saved yet!

Video 4: default initialization

25 Initialization

- Every object in C++ is initialized upon creation
- Objects can be explicitly initialized

```
int main() {
  int x = 5;
  int array1[3] = { 3, 4, 5 };
  Triangle t1(3, 4, 5);
  Triangle t2 = Triangle(3, 4, 5);
}
```

- Objects can also be *default initialized*

```
int main() {
  int y;
  int array2[3];
  Triangle t3;
}
```

26 Default Initialization

- Objects that are not explicitly initialized are **default initialized**
- Atomic objects (`int`, `double`, `bool`, `char`, pointers) are default initialized by doing nothing
  - They retain whatever value was previously there in memory (junk)
- Array objects are default initialized by default initializing each element
- Compound (i.e. class-type) objects are default initialized by calling the default constructor

```
int main() {
  int y;           // contains junk
  int array2[3];   // each element contains
  Triangle t3;     // 1x1x1 equilateral tria
}
```

Exercise: Default Initialization Syntax

27

- Line A creates a default-initialized 1x1x1 triangle.
- What does line B do?

```
class Triangle {
private:
  double a, b, c;

public:
  Triangle()
    : a(1), b(1), c(1) { }
};

int main() {
  A Triangle t1;
  B Triangle t2();
}
```

Question

A) It does the same thing as line A.

B) The ( ) syntax can't be used in declarations, so this doesn't compile.

C) t2 is created as a triangle, but initialized with memory junk.

D) t2 is declared as a function that returns a Triangle.

E) It calls the constructor function, but doesn't c Triangle object.

28 The Implicit Default Constructor

- If you don't define **any** constructors, the compiler provides a default constructor for you.

```
struct Person {
  int age;
  string name;
  bool isNinja;
  // implicit default ctor
  // Person() {}
};
```

- If you define **any** constructors, the compiler **doesn't** give you a default one automatica (And if you don't write it, there is no default

29 Default Initialization of Members

- Members of compound objects are default initialized if not explicitly initialized

```
struct Person {
  int age;
  string name;
  bool isNinja;
  // implicit default ctor
  // Person() {}
};

int main() {
  Person alex;
  Person jon = { 25, "jon", true };
}
```

junk (not necessarily 0)

Members not explicitly initialized

The Stack

main
alex Person
0x1000 age 0
0x1004 name ""
0x1008 isNinja false

jon Person
0x1009 age 25
0x1013 name "jo
0x1017 isNinja

Video 5: Best practices for ADTs with classes

30

`struct` vs. `class`

- In the C++ language, the only difference between the `struct` and `class` keywords is the default access level for members.
  - `struct` – `public` by default
  - `class` – `private` by default

```
struct Triangle {
  double a;
  double b;
  double c;
  ...
};
```
a, b, c are public

```
class Triangle {
  double a;
  double b;
  double c;
  ...
};
```
a, b, c are private

- However, **by convention** we use structs and classes very differently!

31

Member Initializer Lists

- ALWAYS use a member initializer list if you can.

DO
```
class Triangle {
private:
  double a; double b; double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) { }
```

DON'T
```
  Triangle(double a_in, double b_in, double c_in) {
    : a(), b(), c() {
    a = a_in;
    b = b_in;
    c = c_in;
  }
};
```
Compiler sees this as a member initializer list. a, are default-initialized fi assigned values later in

32

Member Initializer Lists

- ALWAYS use a member initializer list if you can.

DO
```
class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;

public:
  Professor(int creams, int sugars)
    : favCoffee(creams, sugars) { }
```

DON'T
```
  Professor(int creams, int sugars) {
    favCoffee = Coffee(creams, sugars);
  }
};
```
Error: Compiler attempts to default cor favCoffee before the body of the c

33

Review: Representation Invariants

- A problem for compound types…
  - Some combinations of member values don't make sense together.
- We use **representation invariants** to express the conditions for a **valid** compound object.
- For Triangle:

Positive Edge Lengths	Triangle Inequality
0 < a	a + b > c
0 < b	a + c > b
0 < c	b + c > a

34

Check Invariants

```
class Triangle {
private:
  double a;
  double b;
  double c;

  void check_invariants() {
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  && a + c > b && b + c > a);
  }

public:
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {
    check_invariants();
  }
};
```
Member function to check invariants

this helper function is created in the private space, which means outside of the class the user will not be allowed to use it. It is special for the public member function to check invariants.

Check invariants any member variables ar

35

Get and Set Functions

- Some classes provide functions to get and set private member variables

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  double get_a() const {
    return a;
  }

  void set_a(double a_in) {
    a = a_in;
    check_invariants();
  }
};
```
the reason why we create a function to get and set the private member variables instead of making them public is that we can control if the value pass in is meaningful.

Check invariants ar member variables

36

Good Abstraction Design

- Encapsulation 封装
  - C++ groups data and behavior together in a class.
  - It gives us mechanisms to protect representation invariants. (access control, constructors)
- Separate **interface** from **implementation**.
  - Work only with the interface, and "hide" away the implementation.
  - Avoid improper dependencies on th implementation.

37

C-Style Information Hiding

Triangle.h
Interface
What a Triangle does.

```
struct Triangle {
  double a, b, c;
};

double Triangle_perimeter(Triangle const *tri);
Void Triangle_scale(Triangle *tri, double s);
```

Triangle.cpp
Implementation
Details of how it does it.

```
#include "Triangle.h"

double Triangle_perimeter(Triangle const *tri) {
  return tri->a + tri->b + tri->c;
}
void Triangle_scale(Triangle *tri, double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

38

Information Hiding in C++

Triangle.h
Interface
- What a Triangle does.

```
class Triangle {
public:
  Triangle();
  Triangle(double a_in,
           double b_in,
           double c_in);
  double area() const;
  double perimeter() const;
  void scale(double s);

private:
  double a, b, c;
};
```
Private members are implementation details, so they should be at the bottom.

Triangle.cpp
Implementation
- Details of how it does it.

```
#include "Triangle.h"

Triangle::Triangle(double a_in,
  double b_in, double c_in)
  : a(a_in), b(b_in), c(c_in) { }

void Triangle::scale(double s) {
  a *= s;
  b *= s;
  c *= s;
}
```
fully qualified — scope resolution operator

The scope resolution operator to refer t function

39

Testing a C++ ADT

```
#include "Triangle.h"
#include "unit_test_framework.h"

TEST(test_triangle_basic) {
  Triangle t(3, 4, 5);
  ASSERT_EQUAL(t.area(), 6);
  ASSERT_EQUAL(t.get_a(), 3);
  t.set_a(4);
  ASSERT_EQUAL(t.get_a(), 4);
}

TEST_MAIN()
```
C++ forces you to respect the interface